3.4 Uninformed Search Strategies An uninformed search algorithm is given no clue about how close a state is to the goal(s). For example, consider our agent in Arad with the goal of reaching Bucharest. An uninformed agent with no knowledge of Romanian geography has no clue whether going to Zerind or Sibiu is a better first step. In contrast, an informed agent (Section 3.5) who knows the location of each city knows that Sibiu is much closer to Bucharest and thus more likely to be on the shortest path. 3.4.1 Breadth-first search When all actions have the same cost, an appropriate strategy is breadth-first search, in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. This is a systematic search strategy that is therefore complete even on infinite state spaces. We could implement breadth-first search as a call to BEST-FIRST-SEARCH where the evaluation function is the depth of the node—that is, the number of actions it takes to reach the node. Breadth-first search However, we can get additional efficiency with a couple of tricks. A first-in-first-out queue will be faster than a priority queue, and will give us the correct order of nodes: new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first. In addition, reached can be a set of states rather than a mapping from states to nodes, because once we've reached a state, we can never find a better path to the state. That also means we can do an early goal test, checking whether a node is a solution as soon as it is generated, rather than the late goal test that best-first search uses, waiting until a node is popped off the queue. Figure 3.8 shows the progress of a breadth-first search on a binary tree, and Figure 3.9 shows the algorithm with the early-goal efficiency enhancements. f(n) Figure 3.8 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker. Figure 3.9 Breadth-first search and uniform-cost search algorithms. Early goal test Late goal testBreadth-first search always finds a solution with a minimal number of actions, because when it is generating nodes at depth it has already generated all the nodes at depth so if one of them were a solution, it would have been found. That means it is cost-optimal for problems where all actions have the same cost, but not for problems that don't have that property. It is complete in either case. In terms of time and space, imagine searching a uniform tree where every state has successors. The root of the search tree generates nodes, each of which generates more nodes, for a total of at the second level. Each of these generates more nodes, yielding nodes at the third level, and so on. Now suppose that the solution is at depth Then the total number of nodes generated is All the nodes remain in memory, so both time and space complexity are Exponential bounds like that are scary. As a typical real-world example, consider a problem with branching factor processing speed 1 million nodes/second, and memory requirements of 1 Kbyte/node. A search to depth would take less than 3 hours, but would require 10 terabytes of memory. The memory requirements are a bigger problem for breadth-first search than the execution time. But time is still an important factor. At depth even with infinite memory, the search would take 3.5 years. In general, exponential complexity search problems cannot be solved by uninformed search for any but the smallest instances. 3.4.2 Dijkstra's algorithm or uniform-cost search When actions have different costs, an obvious choice is to use best-first search where the evaluation function is the cost of the path from the root to the current node. This is called Dijkstra's algorithm by the theoretical computer science community, and uniform-cost search by the AI community. The idea is that while breadth-first search spreads out in waves of uniform depth—first depth 1, then depth 2, and so on—uniform-cost search spreads out in waves of uniform path-cost. The algorithm can be implemented as a call to BEST-FIRST SEARCH with PATH-COST as the evaluation function, as shown in Figure 3.9 . Uniform-cost search d, d − 1, b b b b 2 b b 3 d. 1 + b + b 2 + b 3 + ⋯ + b d = O (b d) O(b d). b = 10, d = 10 d = 14, Consider Figure 3.10 , where the problem is to get from Sibiu to Bucharest. The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively. The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost The least cost node is now Fagaras, so it is expanded, adding Bucharest with

cost Bucharest is the goal, but the algorithm tests for goals only when it expands a node, not when it generates a node, so it has not yet detected that this is a path to the goal. Figure 3.10 Part of the Romania state space, selected to illustrate uniform-cost search. The algorithm continues on, choosing Pitesti for expansion next and adding a second path to Bucharest with cost It has a lower cost, so it replaces the previous path in reached and is added to the frontier. It turns out this node now has the lowest cost, so it is considered next, found to be a goal, and returned. Note that if we had checked for a goal upon generating a node rather than when expanding the lowest-cost node, then we would have returned a higher-cost path (the one through Fagaras). The complexity of uniform-cost search is characterized in terms of the cost of the optimal solution, and a lower bound on the cost of each action, with Then the algorithm's worst-case time and space complexity is which can be much greater than This is because uniform-cost search can explore large trees of actions with low costs before exploring paths involving a high-cost and perhaps useful action. When all action costs are equal, is just and uniform-cost search is similar to breadth-first search. 80 + 97 = 177. 99 + 211 = 310. 80 + 97 + 101 = 278. $C*$, 8 ϵ, $\epsilon > 0$. $O(b^{1+\lfloor C*/\epsilon \rfloor})$, b d . $b^{1+\lfloor C*/\epsilon \rfloor}$ b d+1 ,8 Here, and throughout the book, the "star" in means an optimal value for Uniform-cost search is complete and is cost-optimal, because the first solution it finds will have a cost that is at least as low as the cost of any other node in the frontier. Uniform-cost search considers all paths systematically in order of increasing cost, never getting caught going down a single infinite path (assuming that all action costs are). 3.4.3 Depth-first search and the problem of memory Depth-first search Depth-first search always expands the deepest node in the frontier first. It could be implemented as a call to BEST-FIRST-SEARCH where the evaluation function is the negative of the depth. However, it is usually implemented not as a graph search but as a tree-like search that does not keep a table of reached states. The progress of the search is illustrated in Figure 3.11 ; search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. The search then "backs up" to the next deepest node that still has unexpanded successors. Depth-first search is not cost-optimal; it returns the first solution it finds, even if it is not cheapest. Figure 3.11 $C*$ C. > ϵ > 0 f A dozen steps (left to right, top to bottom) in the progress of a depth-first search on a binary tree from start state A to goal M. The frontier is in green, with a triangle marking the node to be expanded next. Previously expanded nodes are lavender, and potential future nodes have faint dashed lines. Expanded nodes with no descendants in the frontier (very faint lines) can be discarded. For finite state spaces that are trees it is efficient and complete; for acyclic state spaces it may end up expanding the same state many times via different paths, but will (eventually) systematically explore the entire space. In cyclic state spaces it can get stuck in an infinite loop; therefore some implementations of depth-first search check each new node for cycles. Finally, in infinite state spaces, depth-first search is not systematic: it can get stuck going down an infinite path, even if there are no cycles. Thus, depth-first search is incomplete. With all this bad news, why would anyone consider using depth-first search rather than breadth-first or best-first? The answer is that for problems where a tree-like search is feasible, depth-first search has much smaller needs for memory. We don't keep a reachedtable at all, and the frontier is very small: think of the frontier in breadth-first search as the surface of an ever-expanding sphere, while the frontier in depth-first search is just a radius of the sphere. For a finite tree-shaped state-space like the one in Figure 3.11 , a depth-first tree-like search takes time proportional to the number of states, and has memory complexity of only where is the branching factor and is the maximum depth of the tree. Some problems that would require exabytes of memory with breadth-first search can be handled with only kilobytes using depth-first search. Because of its parsimonious use of memory, depth-first tree-like search has been adopted as the basic workhorse of many areas of AI, including constraint satisfaction (Chapter 6), propositional satisfiability (Chapter 7), and logic programming (Chapter 9). A variant of depth-first search called backtracking search uses even less memory. (See Chapter 6 for more details.) In backtracking, only

one successor is generated at a time rather than all successors; each partially expanded node remembers which successor to generate next. In addition, successors are generated by modifying the current state description directly rather than allocating memory for a brand-new state. This reduces the memory requirements to just one state description and a path of actions; a significant savings over states for depth-first search. With backtracking we also have the option of maintaining an efficient set data structure for the states on the current path, allowing us to check for a cyclic path in time rather than For backtracking to work, we must be able to undo each action when we backtrack. Backtracking is critical to the success of many problems with large state descriptions, such as robotic assembly. Backtracking search 3.4.4 Depth-limited and iterative deepening search To keep depth-first search from wandering down an infinite path, we can use depth-limited search, a version of depth-first search in which we supply a depth limit, and treat all nodes at depth as if they had no successors (see Figure 3.12). The time complexity is O(bm), b m   O(m) O(bm) O(1) O(m). ℓ, ℓ and the space complexity is Unfortunately, if we make a poor choice for the algorithm will fail to reach the solution, making it incomplete again. Figure 3.12 Iterative deepening and depth-limited tree-like search. Iterative deepening repeatedly applies depth limited search with increasing limits. It returns one of three different types of values: either a solution node; or failure, when it has exhausted all nodes and proved there is no solution at any depth; or cutoff, to mean there might be a solution at a deeper depth than This is a tree-like search algorithm that does not keep track of reached states, and thus uses much less memory than best-first search, but runs the risk of visiting the same state multiple times on different paths. Also, if the IS-CYCLE check does not check all cycles, then the algorithm may get caught in a loop. Depth-limited search Since depth-first search is a tree-like search, we can’t keep it from wasting time on redundant paths in general, but we can eliminate cycles at the cost of some computation time. If we look only a few links up in the parent chain we can catch most cycles; longer cycles are handled by the depth limit. O(b ℓ) O(bℓ). ℓ ℓ.Sometimes a good depth limit can be chosen based on knowledge of the problem. For example, on the map of Romania there are 20 cities. Therefore, is a valid limit. But if we studied the map carefully, we would discover that any city can be reached from any other city in at most 9 actions. This number, known as the diameter of the state-space graph, gives us a better depth limit, which leads to a more efficient depth-limited search. However, for most problems we will not know a good depth limit until we have solved the problem. Diameter Iterative deepening search solves the problem of picking a good value for by trying all values: first 0, then 1, then 2, and so on—until either a solution is found, or the depth limited search returns the failure value rather than the cutoff value. The algorithm is shown in Figure 3.12 . Iterative deepening combines many of the benefits of depth-first and breadth-first search. Like depth-first search, its memory requirements are modest: when there is a solution, or on finite state spaces with no solution. Like breadth-first search, iterative deepening is optimal for problems where all actions have the same cost, and is complete on finite acyclic state spaces, or on any finite state space when we check nodes for cycles all the way up the path. Iterative deepening search The time complexity is when there is a solution, or when there is none. Each iteration of iterative deepening search generates a new level, in the same way that breadth first search does, but breadth-first does this by storing all nodes in memory, while iterative deepening does it by repeating the previous levels, thereby saving memory at the cost of more time. Figure 3.13 shows four iterations of iterative-deepening search on a binary search tree, where the solution is found on the fourth iteration. ℓ = 19 ℓ O(bd) O(bm) O(b d) O(bm) Figure 3.13 Four iterations of iterative deepening search for goal on a binary tree, with the depth limit varying from 0 to 3. Note the interior nodes form a single path. The triangle marks the node to expand next; green nodes with dark outlines are on the frontier; the very faint nodes provably can’t be part of a solution with this depth limit. Iterative deepening search may seem wasteful because states near the top of the search tree are re-generated multiple times. But for many state spaces,

most of the nodes are in the bottom level, so it does not matter much that the upper levels are repeated. In an iterative deepening search, the nodes on the bottom level (depth) are generated once, those on the next-to-bottom level are generated twice, and so on, up to the children of the root, which are generated times. So the total number of nodes generated in the worst case is M d dwhich gives a time complexity of —asymptotically the same as breadth-first search. For example, if and the numbers are If you are really concerned about the repetition, you can use a hybrid approach that runs breadth-first search until almost all the available memory is consumed, and then runs iterative deepening from all the nodes in the frontier. In general, iterative deepening is the preferred uninformed search method when the search state space is larger than can fit in memory and the depth of the solution is not known. 3.4.5 Bidirectional search The algorithms we have covered so far start at an initial state and can reach any one of multiple possible goal states. An alternative approach called bidirectional search simultaneously searches forward from the initial state and backwards from the goal state(s), hoping that the two searches will meet. The motivation is that is much less than (e.g., 50,000 times less when). Bidirectional search For this to work, we need to keep track of two frontiers and two tables of reached states, and we need to be able to reason backwards: if state is a successor of in the forward direction, then we need to know that is a successor of in the backward direction. We have a solution when the two frontiers collide. 9 In our implementation, the reached data structure supports a query asking whether a given state is a member, and the frontier data structure (a priority queue) does not, so we check for a collision using reached; but conceptually we are asking if the two frontiers have met up. The implementation can be extended to handle multiple goal states by loading the node for each goal state into the backwards frontier and backwards reached table. N(IDS) = (d)b 1 + (d − 1)b 2 + (d − 2)b 3 … + b d , O(b d) b = 10 d = 5, N(IDS) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 N(BFS) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110. b d/2 + b d/2 b d b = d = 10 s' s s s' 9There are many different versions of bidirectional search, just as there are many different unidirectional search algorithms. In this section, we describe bidirectional best-first search. Although there are two separate frontiers, the node to be expanded next is always one with a minimum value of the evaluation function, across either frontier. When the evaluation function is the path cost, we get bidirectional uniform-cost search, and if the cost of the optimal path is then no node with cost will be expanded. This can result in a considerable speedup. The general best-first bidirectional search algorithm is shown in Figure 3.14 . We pass in two versions of the problem and the evaluation function, one in the forward direction (subscript) and one in the backward direction (subscript). When the evaluation function is the path cost, we know that the first solution found will be an optimal solution, but with different evaluation functions that is not necessarily true. Therefore, we keep track of the best solution found so far, and might have to update that several times before the TERMINATED test proves that there is no possible better solution remaining. Figure 3.14 C ∗ , > C ∗ 2 F BBidirectional best-first search keeps two frontiers and two tables of reached states. When a path in one frontier reaches a state that was also reached in the other half of the search, the two paths are joined (by the function JOIN-NODES) to form a solution. The first solution we get is not guaranteed to be the best; the function TERMINATED determines when to stop looking for new solutions. 3.4.6 Comparing uninformed search algorithms Figure 3.15 compares uninformed search algorithms in terms of the four evaluation criteria set forth in Section 3.3.4 . This comparison is for tree-like search versions which don’t check for repeated states. For graph searches which do check, the main differences are that depth-first search is complete for finite state spaces, and the space and time complexities are bounded by the size of the state space (the number of vertices and edges,). Figure 3.15 |V | + |E|Evaluation of search algorithms. is the branching factor; is the maximum depth of the search tree; is the depth of the shallowest solution, or is when there is no solution; is the depth limit. Superscript caveats are as follows: complete if is finite, and the state space either has a solution or is

finite. complete if all action costs are cost-optimal if action costs are all identical; if both directions are breadth-first or uniform-cost. b m d m ℓ 1 b 2 ≥ ε > 0; 3 43.5 Informed (Heuristic) Search Strategies This section shows how an informed search strategy—one that uses domain-specific hints about the location of goals—can find solutions more efficiently than an uninformed strategy. The hints come in the form of a heuristic function, denoted 10 It may seem odd that the heuristic function operates on a node, when all it really needs is the node's state. It is traditional to use rather than to be consistent with the evaluation function and the path cost Informed search Heuristic function For example, in route-finding problems, we can estimate the distance from the current state to a goal by computing the straight-line distance on the map between the two points. We study heuristics and where they come from in more detail in Section 3.6 . 3.5.1 Greedy best-first search Greedy best-first search is a form of best-first search that expands first the node with the lowest value—the node that appears to be closest to the goal—on the grounds that this is likely to lead to a solution quickly. So the evaluation function Greedy best-first search h(n):10 h (n) h (s) f(n) g (n). h(n) = estimated cost of the cheapest path from the state at node n to a goal state. □ h(n) f(n) = h(n).Let us see how this works for route-finding problems in Romania; we use the straight-line distance heuristic, which we will call If the goal is Bucharest, we need to know the straight-line distances to Bucharest, which are shown in Figure 3.16 . For example, Notice that the values of cannot be computed from the problem description itself (that is, the ACTIONS and RESULT functions). Moreover, it takes a certain amount of world knowledge to know that is correlated with actual road distances and is, therefore, a useful heuristic. Figure 3.16 Values of —straight-line distances to Bucharest. Straight-line distance Figure 3.17 shows the progress of a greedy best-first search using to find a path from Arad to Bucharest. The first node to be expanded from Arad will be Sibiu because the heuristic says it is closer to Bucharest than is either Zerind or Timisoara. The next node to be expanded will be Fagaras because it is now closest according to the heuristic. Fagaras in turn generates Bucharest, which is the goal. For this particular problem, greedy best-first search using finds a solution without ever expanding a node that is not on the solution path. The solution it found does not have optimal cost, however: the path via Sibiu and Fagaras to Bucharest is 32 miles longer than the path through Rimnicu Vilcea and Pitesti. This is why the algorithm is called "greedy"—on each iteration it tries to get as close to a goal as it can, but greediness can lead to worse results than being careful. hSLD. □ hSLD(Arad) = 366. hSLD hSLD hSLD □ hSLD hSLDFigure 3.17 Stages in a greedy best-first tree-like search for Bucharest with the straight-line distance heuristic Nodes are labeled with their -values. Greedy best-first graph search is complete in finite state spaces, but not in infinite ones. The worst-case time and space complexity is With a good heuristic function, however, the complexity can be reduced substantially, on certain problems reaching 3.5.2 A* search The most common informed search algorithm is A* search (pronounced "A-star search"), a best-first search that uses the evaluation function hSLD. h O(|V |). O(bm). f(n) = g(n) + h(n)A* search where is the path cost from the initial state to node and is the estimated cost of the shortest path from to a goal state, so we have In Figure 3.18 , we show the progress of an A* search with the goal of reaching Bucharest. The values of are computed from the action costs in Figure 3.1 , and the values of are given in Figure 3.16 . Notice that Bucharest first appears on the frontier at step (e), but it is not selected for expansion (and thus not detected as a solution) because at it is not the lowest-cost node on the frontier—that would be Pitesti, at Another way to say this is that there might be a solution through Pitesti whose cost is as low as 417, so the algorithm will not settle for a solution that costs 450. At step (f), a different path to Bucharest is now the lowest-cost node, at so it is selected and detected as the optimal solution. Figure 3.18 g(n) n, h(n) n f(n) = estimated cost of the best path that continues from n to a goal. □ g □ hSLD □ f = 450 f = 417. f = 418,Stages in an A* search for Bucharest. Nodes are labeled with The values are the straight-line distances to Bucharest taken from Figure 3.16 . Admissible heuristic A* search is complete. Whether A* is cost-optimal depends

on certain properties of the heuristic. A key property is admissibility: an admissible heuristic is one that never overestimates the cost to reach a goal. (An admissible heuristic is therefore optimistic.) With an admissible heuristic, A* is cost-optimal, which we can show with a proof by contradiction. Suppose the optimal path has cost but the algorithm returns a path with cost Then there must be some node which is on the optimal path and is unexpanded (because if all the nodes on the optimal path had been expanded, then we would have returned that optimal solution). So then, using the notation to mean the cost of the optimal path from the start to and to mean the cost of the optimal path from to the nearest goal, we have: 11 Again, assuming all action costs are and the state space either has a solution or is finite. The first and last lines form a contradiction, so the supposition that the algorithm could return a suboptimal path must be wrong—it must be that A* returns only cost-optimal paths. A slightly stronger property is called consistency. A heuristic is consistent if, for every node and every successor of generated by an action we have: f = g + h. h 11 C∗ , C > C∗ . n g ∗(n) n, h ∗(n) n >∈> 0, f (n) > C ∗ (otherwise n would have been expanded) f (n) = g (n) + h (n) (by definition) f (n) = g ∗ (n) + h (n) (because n is on an optimal path) f (n) ≤ g ∗ (n) + h ∗ (n) (because of admissibility, h (n) ≤ h ∗ (n)) f (n) ≤ C ∗ (by definition, C ∗ = g ∗ (n) + h ∗ (n)) h(n) n n ′ n a, h(n) ≤ c(n, a, n ′) + h(n ′).Consistency This is a form of the triangle inequality, which stipulates that a side of a triangle cannot be longer than the sum of the other two sides (see Figure 3.19). An example of a consistent heuristic is the straight-line distance that we used in getting to Bucharest. Figure 3.19 Triangle inequality: If the heuristic is consistent, then the single number will be less than the sum of the cost of the action from to plus the heuristic estimate Triangle inequality Every consistent heuristic is admissible (but not vice versa), so with a consistent heuristic, A* is cost-optimal. In addition, with a consistent heuristic, the first time we reach a state it will be on an optimal path, so we never have to re-add a state to the frontier, and never have to change an entry in reached. But with an inconsistent heuristic, we may end up with multiple paths reaching the same state, and if each new path has a lower path cost than the previous one, then we will end up with multiple nodes for that state in the frontier, costing us both time and space. Because of that, some implementations of A* take care to only enter a state into the frontier once, and if a better path to the state is found, all the successors of the state are updated (which requires that nodes have child pointers as well as parent pointers). These complications have led many implementers to avoid inconsistent heuristics, but Felner et al. (2011) argues that the worst effects rarely happen in practice, and one shouldn't be afraid of inconsistent heuristics. hSLD h h(n) c(n, a, a ′) n n ′ h(n ′).With an inadmissible heuristic, A* may or may not be cost-optimal. Here are two cases where it is: First, if there is even one cost-optimal path on which is admissible for all nodes on the path, then that path will be found, no matter what the heuristic says for states off the path. Second, if the optimal solution has cost and the second-best has cost and if overestimates some costs, but never by more than then A* is guaranteed to return cost-optimal solutions. 3.5.3 Search contours A useful way to visualize a search is to draw contours in the state space, just like the contours in a topographic map. Figure 3.20 shows an example. Inside the contour labeled 400, all nodes have and so on. Then, because A* expands the frontier node of lowest -cost, we can see that an A* search fans out from the start node, adding nodes in concentric bands of increasing -cost. Figure 3.20 Map of Romania showing contours at and with Arad as the start state. Nodes inside a given contour have costs less than or equal to the contour value. h(n) n C ∗ , C2, h(n) C2 − C ∗ , f(n) = g(n) + h(n) ≤ 400, f f f = 380, f = 400, f = 420, f = g + hContour With uniform-cost search, we also have contours, but of -cost, not The contours with uniform-cost search will be "circular" around the start state, spreading out equally in all directions with no preference towards the goal. With A* search using a good heuristic, the bands will stretch toward a goal state (as in Figure 3.20) and become more narrowly focused around an optimal path. It should be clear that as you extend a path, the costs are monotonic: the path cost always increases as you go along a path, because action costs are always positive. Therefore you get concentric

contour lines that don't cross each other, and if you choose to draw the lines fine enough, you can put a line between any two nodes on any path. 12 Technically, we say "strictly monotonic" for costs that always increase, and "monotonic" for costs that never decrease, but might remain the same. Monotonic But it is not obvious whether the cost will monotonically increase. As you extend a path from to the cost goes from to Canceling out the term, we see that the path's cost will be monotonically increasing if and only if in other words if and only if the heuristic is consistent. But note that a path might contribute several nodes in a row with the same score; this will happen whenever the decrease in is exactly equal to the action cost just taken (for example, in a grid problem, when is in the same row as the goal and you take a step towards the goal, is increased by 1 and is decreased by 1). If is the cost of the optimal solution path, then we can say the following: 13 In fact, the term "monotonic heuristic" is a synonym for "consistent heuristic." The two ideas were developed independently, and then it was proved that they are equivalent (Pearl, 1984). g g + h. g + h g 12 f = g + h n n ′ , g (n) + h (n) g(n) + c(n, a, n ′) + h(n ′). g(n) h(n) ≤ c(n, a, n ′) + h(n ′); 13 g(n) + h(n) h n g h C ∗A* expands all nodes that can be reached from the initial state on a path where every node on the path has We say these are surely expanded nodes. Surely expanded nodes A* might then expand some of the nodes right on the "goal contour" (where) before selecting a goal node. A* expands no nodes with We say that A* with a consistent heuristic is optimally efficient in the sense that any algorithm that extends search paths from the initial state, and uses the same heuristic information, must expand all nodes that are surely expanded by A* (because any one of them could have been part of an optimal solution). Among the nodes with one algorithm could get lucky and choose the optimal one first while another algorithm is unlucky; we don't consider this difference in defining optimal efficiency. Optimally efficient A* is efficient because it prunes away search tree nodes that are not necessary for finding an optimal solution. In Figure 3.18(b) we see that Timisoara has and Zerind has Even though they are children of the root and would be among the first nodes expanded by uniform-cost or breadth-first search, they are never expanded by A* search because the solution with is found first. The concept of pruning—eliminating possibilities from consideration without having to examine them—is important for many areas of AI. Pruning f(n) < C ∗ . f(n) = C ∗ f(n) > C ∗ . f(n) = C ∗ , f = 447 f = 449. f = 418That A* search is complete, cost-optimal, and optimally efficient among all such algorithms is rather satisfying. Unfortunately, it does not mean that A* is the answer to all our searching needs. The catch is that for many problems, the number of nodes expanded can be exponential in the length of the solution. For example, consider a version of the vacuum world with a super-powerful vacuum that can clean up any one square at a cost of 1 unit, without even having to visit the square; in that scenario, squares can be cleaned in any order. With initially dirty squares, there are states where some subset has been cleaned; all of those states are on an optimal solution path, and hence satisfy so all of them would be visited by A*. 3.5.4 Satisficing search: Inadmissible heuristics and weighted A* Inadmissible heuristic A* search has many good qualities, but it expands a lot of nodes. We can explore fewer nodes (taking less time and space) if we are willing to accept solutions that are suboptimal, but are "good enough"—what we call satisficing solutions. If we allow A* search to use an inadmissible heuristic—one that may overestimate—then we risk missing the optimal solution, but the heuristic can potentially be more accurate, thereby reducing the number of nodes expanded. For example, road engineers know the concept of a detour index, which is a multiplier applied to the straight-line distance to account for the typical curvature of roads. A detour index of 1.3 means that if two cities are 10 miles apart in straight-line distance, a good estimate of the best path between them is 13 miles. For most localities, the detour index ranges between 1.2 and 1.6. Detour index N 2N f(n) < C ∗ ,We can apply this idea to any problem, not just ones involving roads, with an approach called weighted A* search where we weight the heuristic value more heavily, giving us the evaluation function for some Weighted A* search Figure 3.21 shows a search problem on a grid world. In (a), an A* search finds

the optimal solution, but has to explore a large portion of the state space to find it. In (b), a weighted A* search finds a solution that is slightly costlier, but the search time is much faster. We see that the weighted search focuses the contour of reached states towards a goal. That means that fewer states are explored, but if the optimal path ever strays outside of the weighted search's contour (as it does in this case), then the optimal path will not be found. In general, if the optimal solution costs a weighted A* search will find a solution that costs somewhere between and but in practice we usually get results much closer to than Figure 3.21 Two searches on the same grid: (a) an A* search and (b) a weighted A* search with weight The gray bars are obstacles, the purple line is the path from the green start to red goal, and the small dots are states that were reached by each search. On this particular problem, weighted A* explores 7 times fewer states and finds a path that is 5% more costly. f(n) = g(n) + W × h(n), W > 1. C ∗ , C ∗ W × C ∗ ; C ∗ W × C ∗ . W = 2.We have considered searches that evaluate states by combining and in various ways; weighted A* can be seen as a generalization of the others: You could call weighted A* "somewhat-greedy search": like greedy best-first search, it focuses the search towards a goal; on the other hand, it won't ignore the path cost completely, and will suspend a path that is making little progress at great cost. There are a variety of suboptimal search algorithms, which can be characterized by the criteria for what counts as "good enough." In bounded suboptimal search, we look for a solution that is guaranteed to be within a constant factor of the optimal cost. Weighted A* provides this guarantee. In bounded-cost search, we look for a solution whose cost is less than some constant And in unbounded-cost search, we accept a solution of any cost, as long as we can find it quickly. Bounded suboptimal search Bounded-cost search Unbounded-cost search An example of an unbounded-cost search algorithm is speedy search, which is a version of greedy best-first search that uses as a heuristic the estimated number of actions required to reach a goal, regardless of the cost of those actions. Thus, for problems where all actions g h A* search: g(n) + h(n) (W = 1) Uniform-cost search: g(n) (W = 0) Greedy best-first search: h(n) (W = ∞) Weighted A* search: g(n) + W × h(n) (1 < W < ∞) W C.have the same cost it is the same as greedy best-first search, but when actions have different costs, it tends to lead the search to find a solution quickly, even if it might have a high cost. Speedy search 3.5.5 Memory-bounded search The main issue with A* is its use of memory. In this section we'll cover some implementation tricks that save space, and then some entirely new algorithms that take better advantage of the available space. Memory is split between the frontier and the reached states. In our implementation of best first search, a state that is on the frontier is stored in two places: as a node in the frontier (so we can decide what to expand next) and as an entry in the table of reached states (so we know if we have visited the state before). For many problems (such as exploring a grid), this duplication is not a concern, because the size of frontier is much smaller than reached, so duplicating the states in the frontier requires a comparatively trivial amount of memory. But some implementations keep a state in only one of the two places, saving a bit of space at the cost of complicating (and perhaps slowing down) the algorithm. Another possibility is to remove states from reached when we can prove that they are no longer needed. For some problems, we can use the separation property (Figure 3.6 on page 72), along with the prohibition of U-turn actions, to ensure that all actions either move outwards from the frontier or onto another frontier state. In that case, we need only check the frontier for redundant paths, and we can eliminate the reached table. For other problems, we can keep reference counts of the number of times a state has been reached, and remove it from the reached table when there are no more ways to reach the state. For example, on a grid world where each state can be reached only from its four neighbors, once we have reached a state four times, we can remove it from the table. Reference count Now let's consider new algorithms that are designed to conserve memory usage. Beam search limits the size of the frontier. The easiest approach is to keep only the nodes with the best -scores, discarding any other expanded nodes. This of course makes the search incomplete and suboptimal, but we can choose to make

good use of available memory, and the algorithm executes fast because it expands fewer nodes. For many problems it can find good near-optimal solutions. You can think of uniform-cost or A* search as spreading out everywhere in concentric contours, and think of beam search as exploring only a focused portion of those contours, the portion that contains the best candidates. Beam search An alternative version of beam search doesn't keep a strict limit on the size of the frontier but instead keeps every node whose -score is within of the best -score. That way, when there are a few strong-scoring nodes only a few will be kept, but if there are no strong nodes then more will be kept until a strong one emerges. Iterative-deepening A* search (IDA*) is to A* what iterative-deepening search is to depth first: IDA* gives us the benefits of A* without the requirement to keep all reached states in memory, at a cost of visiting some states multiple times. It is a very important and commonly used algorithm for problems that do not fit in memory. Iterative-deepening A* search k f k k f δ fIn standard iterative deepening the cutoff is the depth, which is increased by one each iteration. In IDA* the cutoff is the -cost ( ); at each iteration, the cutoff value is the smallest -cost of any node that exceeded the cutoff on the previous iteration. In other words, each iteration exhaustively searches an -contour, finds a node just beyond that contour, and uses that node's -cost as the next contour. For problems like the 8-puzzle where each path's -cost is an integer, this works very well, resulting in steady progress towards the goal each iteration. If the optimal solution has cost then there can be no more than iterations (for example, no more than 31 iterations on the hardest 8-puzzle problems). But for a problem where every node has a different -cost, each new contour might contain only one new node, and the number of iterations could be equal to the number of states. Recursive best-first search (RBFS) (Figure 3.22) attempts to mimic the operation of standard best-first search, but using only linear space. RBFS resembles a recursive depth first search, but rather than continuing indefinitely down the current path, it uses the _limit variable to keep track of the -value of the best alternative path available from any ancestor of the current node. If the current node exceeds this limit, the recursion unwinds back to the alternative path. As the recursion unwinds, RBFS replaces the -value of each node along the path with a backed-up value—the best -value of its children. In this way, RBFS remembers the -value of the best leaf in the forgotten subtree and can therefore decide whether it's worth reexpanding the subtree at some later time. Figure 3.23 shows how RBFS reaches Bucharest. Figure 3.22 f g + h f f f f C ∗ , C ∗ f f f f f fThe algorithm for recursive best-first search. Figure 3.23Stages in an RBFS search for the shortest route to Bucharest. The -limit value for each recursive call is shown on top of each current node, and every node is labeled with its -cost. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras). (b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea; then Fagaras is expanded, revealing a best leaf value of 450. (c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed up to Fagaras; then Rimnicu Vilcea is expanded. This time, because the best alternative path (through Timisoara) costs at least 447, the expansion continues to Bucharest. Recursive best-first search f fBacked-up value RBFS is somewhat more efficient than IDA*, but still suffers from excessive node regeneration. In the example in Figure 3.23 , RBFS follows the path via Rimnicu Vilcea, then "changes its mind" and tries Fagaras, and then changes its mind back again. These mind changes occur because every time the current best path is extended, its -value is likely to increase— is usually less optimistic for nodes closer to a goal. When this happens, the second-best path might become the best path, so the search has to backtrack to follow it. Each mind change corresponds to an iteration of IDA* and could require many reexpansions of forgotten nodes to recreate the best path and extend it one more node. RBFS is optimal if the heuristic function is admissible. Its space complexity is linear in the depth of the deepest optimal solution, but its time complexity is rather difficult to characterize: it depends both on the accuracy of the heuristic function and on how often the best path changes as nodes are expanded. It expands nodes in order

of increasing -score, even if is nonmonotonic. IDA* and RBFS suffer from using too little memory. Between iterations, IDA* retains only a single number: the current -cost limit. RBFS retains more information in memory, but it uses only linear space: even if more memory were available, RBFS has no way to make use of it. Because they forget most of what they have done, both algorithms may end up reexploring the same states many times over. It seems sensible, therefore, to determine how much memory we have available, and allow an algorithm to use all of it. Two algorithms that do this are MA* (memory-bounded A*) and SMA* (simplified MA*). SMA* is—well—simpler, so we will describe it. SMA* proceeds just like A*, expanding the best leaf until memory is full. At this point, it cannot add a new node to the search tree without dropping an old one. SMA* always drops the worst leaf node—the one with the highest -value. Like RBFS, SMA* then backs up the value of the forgotten node to its parent. In this way, the ancestor of a forgotten subtree knows the quality of the best path in that subtree. With this information, SMA* regenerates the subtree only when all other paths have been shown to look worse than the path it has forgotten. Another way of saying this is that if all the descendants of a node are forgotten, then we f h h(n) f f f f nwill not know which way to go from but we will still have an idea of how worthwhile it is to go anywhere from MA* SMA* The complete algorithm is described in the online code repository accompanying this book. There is one subtlety worth mentioning. We said that SMA* expands the best leaf and deletes the worst leaf. What if all the leaf nodes have the same -value? To avoid selecting the same node for deletion and expansion, SMA* expands the newest best leaf and deletes the oldest worst leaf. These coincide when there is only one leaf, but in that case, the current search tree must be a single path from root to leaf that fills all of memory. If the leaf is not a goal node, then even if it is on an optimal solution path, that solution is not reachable with the available memory. Therefore, the node can be discarded exactly as if it had no successors. SMA* is complete if there is any reachable solution—that is, if the depth of the shallowest goal node, is less than the memory size (expressed in nodes). It is optimal if any optimal solution is reachable; otherwise, it returns the best reachable solution. In practical terms, SMA* is a fairly robust choice for finding optimal solutions, particularly when the state space is a graph, action costs are not uniform, and node generation is expensive compared to the overhead of maintaining the frontier and the reached set. On very hard problems, however, it will often be the case that SMA* is forced to switch back and forth continually among many candidate solution paths, only a small subset of which can fit in memory. (This resembles the problem of thrashing in disk paging systems.) Then the extra time required for repeated regeneration of the same nodes means that problems that would be practically solvable by A*, given unlimited memory, become intractable for SMA*. That is to say, memory limitations can make a problem intractable from the point of view of computation time. Although no current theory explains the tradeoff n, n. f d,between time and memory, it seems that this is an inescapable problem. The only way out is to drop the optimality requirement. Thrashing 3.5.6 Bidirectional heuristic search With unidirectional best-first search, we saw that using as the evaluation function gives us an A* search that is guaranteed to find optimal-cost solutions (assuming an admissible) while being optimally efficient in the number of nodes expanded. With bidirectional best-first search we could also try using but unfortunately there is no guarantee that this would lead to an optimal-cost solution, nor that it would be optimally efficient, even with an admissible heuristic. With bidirectional search, it turns out that it is not individual nodes but rather pairs of nodes (one from each frontier) that can be proved to be surely expanded, so any proof of efficiency will have to consider pairs of nodes (Eckerle et al., 2017). We'll start with some new notation. We use for nodes going in the forward direction (with the initial state as root) and for nodes in the backward direction (with a goal state as root). Although both forward and backward searches are solving the same problem, they have different evaluation functions because, for example, the heuristics are different depending on whether you are striving for the goal or for the initial state. We'll assume admissible

heuristics. Consider a forward path from the initial state to a node and a backward path from the goal to a node We can define a lower bound on the cost of a solution that follows the path from the initial state to then somehow gets to then follows the path to the goal as In other words, the cost of such a path must be at least as large as the sum of the path costs of the two parts (because the remaining connection between them must have nonnegative f(n) = g(n) + h(n) h f(n) = g(n) + h(n), fF (n) = gF (n) + hF (n) fB(n) = gB(n) + hB(n) m n. m, n, lb(m, n) = max(gF (m) + gB(n), fF (m), fB(n))cost), and the cost must also be at least as much as the estimated cost of either part (because the heuristic estimates are optimistic). Given that, the theorem is that for any pair of nodes with less than the optimal cost we must expand either or because the path that goes through both of them is a potential optimal solution. The difficulty is that we don't know for sure which node is best to expand, and therefore no bidirectional search algorithm can be guaranteed to be optimally efficient—any algorithm might expand up to twice the minimum number of nodes if it always chooses the wrong member of a pair to expand first. Some bidirectional heuristic search algorithms explicitly manage a queue of pairs, but we will stick with bidirectional best-first search (Figure 3.14), which has two frontier priority queues, and give it an evaluation function that mimics the criteria: The node to expand next will be the one that minimizes this value; the node can come from either frontier. This function guarantees that we will never expand a node (from either frontier) with We say the two halves of the search "meet in the middle" in the sense that when the two frontiers touch, no node inside of either frontier has a path cost greater than the bound Figure 3.24 works through an example bidirectional search. Figure 3.24 Bidirectional search maintains two frontiers: on the left, nodes A and B are successors of Start; on the right, node F is an inverse successor of Goal. Each node is labeled with values and the value. (The values are the sum of the action costs as shown on each arrow; the values are arbitrary and cannot be derived from anything in the figure.) The optimal solution, Start-A-F Goal, has cost so that means that a meet-in-the-middle bidirectional algorithm should not expand any node with and indeed the next node to be expanded would be A or F (each with), leading us to an optimal solution. If we expanded the node with lowest cost first, then B and C would come next, and D and E would be tied with A, but they all have and thus are never expanded when is the evaluation function. f m, n lb(m, n) C ∗ , m n, (m, n) lb f2(n) = max(2g(n), g(n) + h(n)) f2 f2 g(n) > . C ∗ 2 . C ∗ 2 f = g + h f2 = max(2g, g + h) g h C∗ = 4 + 2 + 4 = 10, g > = 5; C ∗ 2 g = 4 f g > C ∗ 2 f2Front-to-end Front-to-front We have described an approach where the heuristic estimates the distance to the goal (or, when the problem has multiple goal states, the distance to the closest goal) and estimates the distance to the start. This is called a front-to-end search. An alternative, called front-to-front search, attempts to estimate the distance to the other frontier. Clearly, if a frontier has millions of nodes, it would be inefficient to apply the heuristic function to every one of them and take the minimum. But it can work to sample a few nodes from the frontier. In certain specific problem domains it is possible to summarize the frontier—for example, in a grid search problem, we can incrementally compute a bounding box of the frontier, and use as a heuristic the distance to the bounding box. Bidirectional search is sometimes more efficient than unidirectional search, sometimes not. In general, if we have a very good heuristic, then A* search produces search contours that are focused on the goal, and adding bidirectional search does not help much. With an average heuristic, bidirectional search that meets in the middle tends to expand fewer nodes and is preferred. In the worst case of a poor heuristic, the search is no longer focused on the goal, and bidirectional search has the same asymptotic complexity as A*. Bidirectional search with the evaluation function and an admissible heuristic is complete and optimal.